

# RANS-MP: A Portable Parallel Navier-Stokes Solver\*

Rob F. Van der Wijngaart, MRJ Technology Solutions, MS 258-1

Maurice Yarrow, Sterling Software, MS T27A-1

## 1 Introduction

The emerging trend in practical parallel supercomputing in the last few years has been towards a small set of hardware systems and software models. Typical configurations now consist of cache-based distributed, shared or distributed/shared memory machines running one of the two message-passing libraries PVM (Parallel Virtual Machine) or MPI (Message Passing Interface). Increasingly, the scientific community is adopting the more comprehensive and better performing MPI over PVM, and it is expected that MPI will become the industry standard for message-passing programs before long.

As a result, it is now becoming possible to write parallel programs that are *functionally* portable thanks to a common software standard, and that are *effectively* portable thanks to a common hardware standard. Until now, however, parallel platforms have lacked a common standard for I/O, which has limited the true portability of useful engineering programs. This situation stands to be remedied by the introduction of an I/O extension to MPI, a draft of which has been made available recently by the MPI committee [1].

In this paper we describe the implementation and performance of the single-grid compressible flow solver program RANS-MP, which is based on the well-know OVERFLOW and POVERFLOW codes. RANS-MP, which retains the implicit nature of the diagonalized Beam-Warming approximate-factorization scheme underlying the ADI algorithm used in OVERFLOW, makes use of both MPI and a preliminary version of its I/O extension to arrive at a truly portable parallel Navier-Stokes solver. The double-precision code is tuned for general cache-based architectures, but contains no other machine-specific optimizations.

## 2 Algorithm

The domain decomposition method used for implementing the ADI algorithm is the successful three-dimensional multi-partition method (Refs. [2, 3, 4]). We briefly review a two-dimensional version of the method to explain how it can serve to produce a solution algorithm with low communication overhead and a nearly perfect load balance. Figure 1 shows an example of how the computational grid is distributed among 4 processors. Each processor receives multiple grid sub-blocks (of equal shading) that are arranged in such a fashion that during every phase of the ADI line solvers in the x- and y-direction each processor has work

---

\*This work was supported in part by NASA Ames Research Center under contract numbers NAS 2-14303 and NAS 2-13210

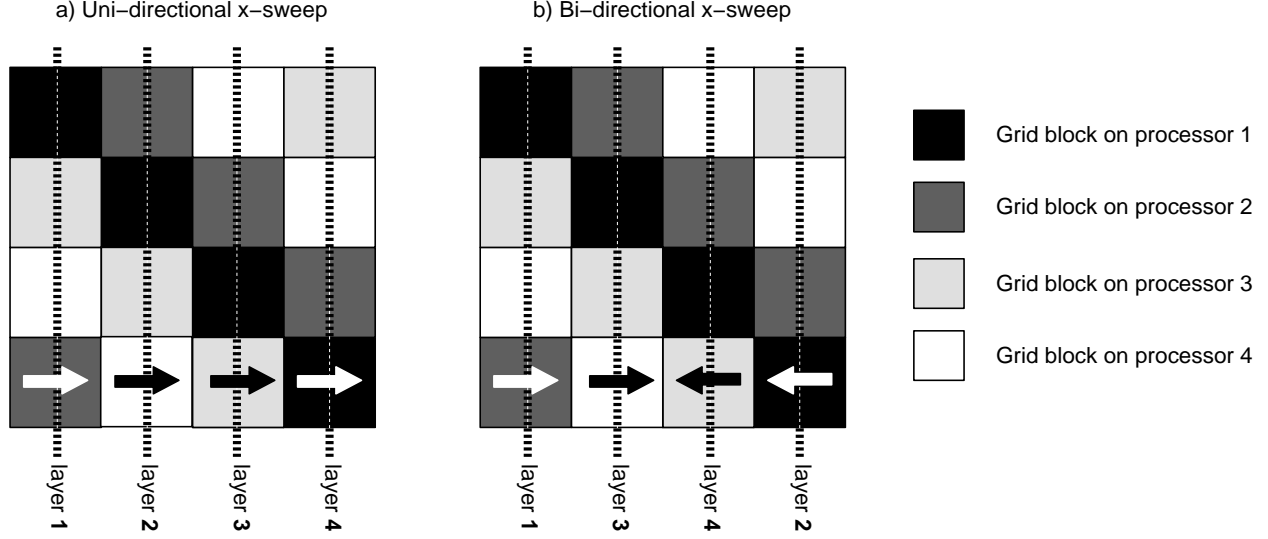


Figure 1: Schematic of x-solves using uni-directional and bi-directional multi-partitioning

to do. In the uni-directional multi-partition method (UMP) the line solvers are started at the beginning of every grid line and progress layer by layer from left to right during the forward elimination process of the x-solve, as in Figure 1a. Moving between layers, processors need to send data to their neighbors in the positive x-direction, and solution along grid lines can only resume after that data has been received; no overlap of computation and communication is possible. In the bi-directional multi-partition (BMP) method the ADI line solves are started on opposite sides of the grid lines, and the solution process works its way towards the middle of the grid layer by layer in an alternating fashion from left to right and from right to left, respectively, as in Figure 1b. Since data is again passed to the nearest neighbor in the line solve direction but is not yet needed until the solution process returns from the other side of the grid line, an effective overlap of computation and communication is obtained. An additional advantage of this overlap is a looser synchronization between the parallel processes, allowing temporary load imbalances due to slight differences in the size of the grid sub-blocks to be hidden. As a general rule we find that the best performance from message-passing programs is obtained when synchronization—both explicit and implicit—is avoided as much as possible.

### 3 Code optimizations

Besides the algorithmic improvement of the code through the use of BMP, the following optimizations were applied:

- Elimination of non-unit-stride array access through loop restructuring and introduction of auxiliary arrays (no additional space was claimed, but temporary workspace was efficiently reused).
- Elimination of many redundant operations by saving intermediate results (again, no additional space was claimed).

- Streamlining of the application of boundary conditions through preprocessing. Some of the boundary conditions require significant logic to determine which processors need to communicate nonlocal data (e.g. C-grid conditions), and all conditions require some logic to determine what subset of the boundary conditions to apply to which grid sub-block. The ‘raw’ boundary conditions supplied through an input file are dissected into a set of ‘atomic’ boundary conditions, each of which pertains to exactly one grid sub-block assigned to a processor. In addition, each atomic condition requires exactly one communication, or none at all. These atomic conditions can be applied blindly during the course of the time stepping in the program.
- Speedup of I/O through the use of “collective” read and write statements from the MPI-IO library [1]. Ideally, parallel CFD codes read for each grid a single input file and write a single output file that represent begin and end states of the program (for restarts and analysis), without regard for the number of processors used in the computation. Because the arrays that make up the solutions to be read and written are spread over multiple processors, the requirement of producing single files typically results in very many disk accesses by the individual processors. The granularity of the I/O can be increased substantially by rearranging among the nodes the data to be written to disk prior to actual output. This is accomplished by using the collective write command `mpio_write_all`. This operation requires the specification of the layout of the data to be written by the nodes in terms of MPI derived data types, but its use is otherwise completely transparent to the user.

## 4 Results on IBM SP2

While the correctness of RANS-MP has been demonstrated for a realistic super-critical-wing design, we adopt a simplified, elliptic, algebraically-generated C-grid to accommodate easy scale-up for presenting performance results. The topology and boundary conditions, however, are kept the same as those for the super-critical wing. Viscous effects in the direction perpendicular to the wing surface are taken into account, but no turbulence model has been implemented yet. All runs are done on the IBM SP2 system installed at the NAS facility at NASA Ames research center. It contains 160 “wide” nodes, each with at least 128 Mbytes of RAM and running AIX 4.1.3.

Two scalability studies are carried out, using the IBM proprietary version of MPI. For the first set of computations the total grid size is kept constant at  $120 \times 45 \times 45$  points, which is the largest size that can be run completely in core on a single processor (the largest dimension is the C-grid wrap-around direction). For the second set, measuring scaled speedup, the number of points per processor is kept constant at approximately 200,000. This is equivalent to a grid of  $100 \times 45 \times 45$  points on a single processor, which is 20% smaller than in the unscaled-speedup case. The 20% difference is necessary to allow for additional memory overhead when scaling up to large numbers of processors. Figures 2 and 3 show the results for the unscaled and scaled program performance, respectively, in millions of points updated per second versus number of processors employed. The performance numbers were obtained by selecting the fastest execution times from six independent runs for each case, in order

to reduce the noise due to system jobs running on the nodes of the SP2 concurrently with the parallel flow solver. Evidently, BMP outperforms UMP in the fixed-grid-size case; The differences run up to 12% on 121 processors. Observe that unscaled parallel efficiency of BMP is still approximately 50% on 81 processors, and that performance degrades gracefully beyond. Scalability of UMP for the fixed grid size lags behind that of BMP, but not by much. It is expected that the difference between UMP and BMP will be larger on loosely coupled networks, such as production workstation clusters, and on other systems that allow more effective overlap between computation and communication than does the NAS IBM SP2. An example of such a system would be the EM-X architecture studied by Sohn et al. [5].

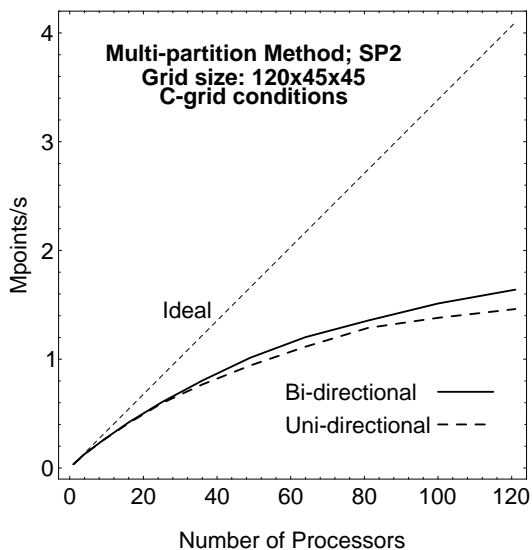


Figure 2: Unscaled speedup

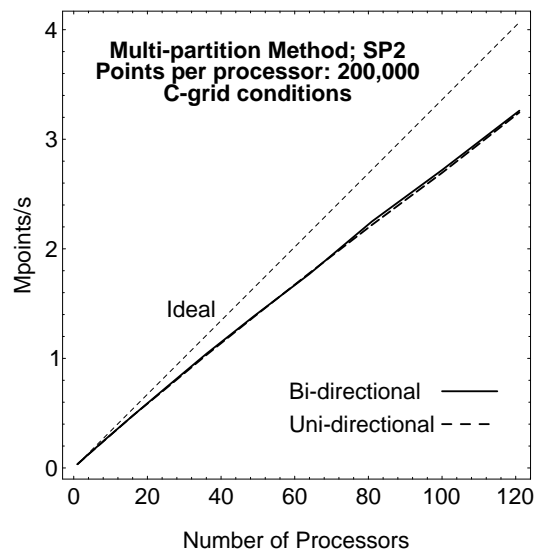


Figure 3: Scaled speedup

When the number of grid points is allowed to scale with the number of processors used, scalability of both UMP and BMP is excellent, as is evidenced by Figure 3. On 121 processors the parallel efficiency is still more than 80%. The distinction between the performances of UMP and BMP is negligible, due to the fact that the communication requirements grow only very slowly with the number of processors.

Besides total performance of the code we also measured the fraction of time spent in the boundary condition routines, since some of these require communications (C-grid) and could create a load imbalance. It was found, however, that this fraction increased only from 0.28% to 1.68% when increasing the number of processors from 1 to 121 while keeping the grid size fixed. Consequently, the boundary conditions never become a performance bottleneck.

A potentially more costly operation is I/O. On many systems without specific hardware and software support, the bandwidth of parallel I/O actually decreases as the number of processors participating in the computation increases. The reason for this is twofold. First, the number of processors connected to the I/O system (I/O server nodes) is often fixed, limiting the absolute aggregate bandwidth attainable by any application, regardless of the number of compute nodes employed. Second, since the data to be written to the output

device is usually scattered among many processors, I/O granularity is often very fine, as was argued in Section 3. It is the latter problem that is addressed by the collective I/O operations provided by the MPI-IO library [1]. The idea behind collective I/O is that bandwidth between processors on a parallel systems is often much higher than between the I/O servers and the file system. Consequently, it pays to assemble contiguous I/O data segments on processors by message passing before attempting the actual write operation. The trick is to make this complex data permutation transparent to the user, which is exactly what is accomplished by MPI-IO. An additional advantage of the collective approach is that it enables architectural optimizations by the MPI-IO implementors, such as number of I/O servers used, or size of buffers employed.

We measured the bandwidth when writing the 4.86 MB solution of the fixed-grid case to disk, both with and without collective I/O. The results are presented in Figure 4. Collective I/O for this case is a factor of 30 faster on average than non-collective I/O, with relative gains increasing with growing numbers of processors. It should be noted that the experimental implementation of MPI-IO at the NAS facility—which currently uses the MPICH public-domain version of MPI—has not been tuned at all for the IBM hardware on which it is being run, so it is expected that the factor of 30 is a lower bound for the relative performance improvement obtained by using collective I/O.

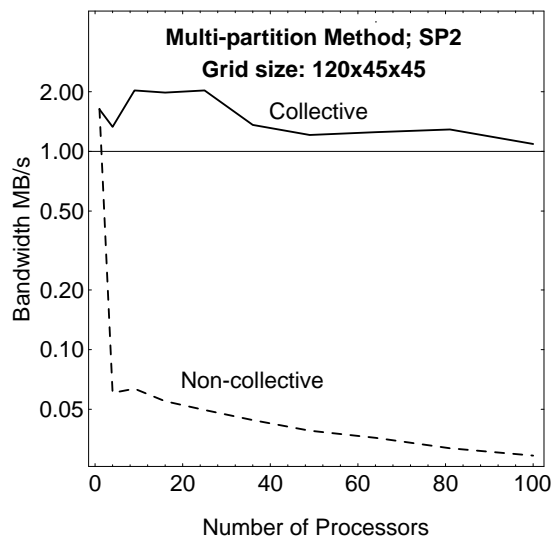


Figure 4: I/O bandwidth for  $120 \times 45 \times 45$  grid

## 5 Discussion and conclusions

Previous reports on UMP by other authors ranged from positive (Kominsky [4]) to rather negative (Naik [6]). Both Kominsky and Naik et al. implemented ARC-3D, which is a precursor to OVERFLOW. Comparing these implementations with each other and with RANS-MP is difficult. The optimistic reports by Kominsky were obtained for a single-precision version of the code on an NCube hypercube, while the generally pessimistic double-precision results by Naik were obtained on the experimental IBM Victor architecture. Victor's host system posed "certain difficulties" [6] at the time the UMP results were obtained that limited scalability tests and necessitated a suboptimal two-dimensional version of the multi-partition method. The applications used to test the codes were also different, Kominsky's flat plate-juncture being markedly simpler than Naik's body of revolution or our wing segment. Both the NCube and Victor had much less memory per node than the IBM SP2 (which significantly limited the maximum possible problem size), and many of their architectural features are now obsolete. Nonetheless, we conclude that our experiments confirm the preliminary

results obtained by Kominsky, namely that a carefully implemented message-passing version of OVERFLOW employing multi-partitioning can exhibit excellent relative and absolute performance on a distributed-memory parallel computer. In addition, it is now possible to transport both the computational and the I/O performance to other machines without recoding.

Finally, we firmly dismiss the criticism that the multi-partition method is significantly more difficult to implement in engineering application programs than the more generally accepted uni-partition methods. While it is true that the somewhat more efficient BMP method adds considerable complexity to the line solver part of the code compared to UMP, the latter method is just as easy (or as hard, depending on the programmer's point of view) to implement as a uni-partition method. Moreover, if optimal performance is demanded from a uni-partition method with (bi-directional) pipelined Gaussian elimination, then a large amount of extra logic is required to cover the special cases of one or two processor grid-blocks in a certain coordinate direction, as well as to determine the grouping size for optimal pipeline granularity. By contrast, BMP scales smoothly and efficiently across a large range of numbers of processors without any special-case coding.

## References

- [1] MPI-IO Committee, *MPI-IO: A parallel File I/O interface for MPI, Version 0.5*, <http://parallel.nas.nasa.gov/MPI-IO/mpi-io.html>
- [2] R.F. Van der Wijngaart, *Efficient implementation of a 3-dimensional ADI method on the iPSC/860*, Supercomputing '93, Portland, OR, November 15-19, 1993
- [3] M.H. Smith, R.F. Van der Wijngaart, M. Yarrow, *Improved multi-partition method for line-based iteration schemes*, Computational Aerosciences Workshop 95, NASA Ames Research Center, Moffett Field, CA, March 7-9, 1995
- [4] P.J. Kominsky, *Performance analysis of an implementation of the Beam and Warming implicit factored scheme on the NCube hypercube*, Proc. Third Symposium on the Frontiers of Massively Parallel Computation, College Park, MD, October 8-10, 1990, IEEE Computer Society Press, Los Alamitos, CA
- [5] A. Sohn et al., *Identifying the capability of overlapping computation with communication*, to appear in Proc. Parallel Architectures and Compilation Techniques, ACM/IEEE, Boston, MA, October 1996
- [6] N.H. Naik, V.K. Naik, M. Nicoules, *Parallelization of a class of implicit finite difference schemes in computational fluid dynamics*, Int. J.High Speed Computing, Vol. 5, No. 1, 1993